Component Architecture of the Tecolote Framework

Mark Zander[1], John Hall[1], Jim Painter[1], Sean O'Rourke[1]

Los Alamos National Laboratory

Abstract. Tecolote is a framework of physics and computer science code written to support the Blanca Project, a part of the Department of Energy's Accelerated Strategic Computing Initiative. The goal of the project is to provide a stable code development environment in the face of changing hardware and software technology. Tecolote relies on another toolkit (POOMA) to provide a consistent interface to different parallel architectures. Tecolote itself aims to facilitate the replacement and reuse of individual software components within complex simulations. This paper focuses on the techniques used by Tecolote to implement this support.

1 Introduction

The Blanca project is part of the Department of Energy's Accelerated Strategic Computing Initiative (ASCI), which focuses on science-based nuclear weapons stockpile stewardship through the large-scale simulation of multi-physics, multi-dimensional, stockpile-relevant problems. Blanca is the only Los Alamos National Laboratory ASCI project written entirely in C++. Tecolote, the underlying framework for the development of Blanca physics codes, provides an infrastructure for combining individual component modules to create large-scale applications encompassing a wide variety of physics models, numerical solution options, and underlying data storage schemes, activating only essential components at run-time [2]. Tecolote has been designed to maximize code re-use and to separate physics from computer science as much as possible, allowing physics model developers to use POOMA to write algorithms in a style similar to the problem's underlying computational physics equations [3].

Tecolote is portable to all ASCI-relevant hardware, making full use of available parallelism. It supports the rapid implementation of physics models and their immediate application to problems on the ASCI scale, providing a powerful, flexible, run-time environment that allows users to create and compose physics codes with various capabilities "on the fly."

The Tecolote framework is layered on the Parallel Object-Oriented Methods and Applications (POOMA) framework [3]. This framework contains architecture and parallelism abstractions that allow the user to write parallel physics codes within the Tecolote framework without worrying about the underlying architecture or communications libraries. POOMA provides C++ fields similar to

Fortran-90 arrays, but with additional features including domain decomposition, load balancing, communications, and compact data storage. POOMA's unique capabilities provide the methods developer with powerful tools for expressing various mesh types and multiple dimensions; this allows application developers to write mesh- and dimension-independent physics code whenever possible. Combined with the unique design of the underlying Blanca software infrastructure, this allows us to keep pace with the ever-changing ASCI environment, rapidly prototype ideas, and build on what others have done, rather than using valuable time to reimplement the same basic models on different architectures.

2 Approach

The Tecolote Framework supplies an application programmer interface (API) that supports factorization of applications into components. It is advantageous to factorize an application to cleanly separate interfaces from implementations, separate conceptually independent subparts of a program, avoid code duplication, and maximize code reuse. After factorization, the user may integrate desired components using techniques supplied by Tecolote. Key concepts supported by components include separation of computer science from physics in simulations, implementation independence of component interfaces, and increased run-time configurability (through an input-file scripting language). This flexible approach is made possible through the use of C++ inheritance and virtual function polymorphism. Not surprisingly, a Tecolote component's increased modularity comes at a price. Creation of and communication between components can be somewhat more expensive than the corresponding operations on ordinary C++ objects. Thus, the granularity of a component must be large enough that these operations do not impact the efficiency of the code.

Tecolote facilitates factorization through several mechanisms:

1. a uniform run-time data-sharing interface (the DataDirectory) with dynamic scoping rules;
2. a facility for the run-time description of a component's type and inheritance relations (its MetaType) and the registration of this information in a single type table (the MetaSet); and
3. a means of configuring both data values and control flow without recompilation (the input file scripting language).

In addition to these features, Tecolote supports the separation of computation and I/O through the use of "persistents." By explicitly designating data within a component class as "persistents," the user may indicate what parts of a component will be available to Tecolote's generic I/O modules. This separation of data manipulation from I/O increases component reusability in the context of changing I/O formats.

In the remainder of the paper we illustrate the above-described mechanisms and their interaction through the extended example of a gamma-law equation of state (EOS) model.

3 Setting Component Parameters - Persistents

In object-oriented programming, I/O methods are generally encapsulated in application classes. If one wants to adopt a new I/O format, whether that be binary instead of ASCII text or eight-digit floating-point output instead of six-digit, every application class will need to be modified.

In Tecolote, we separate what is needed for I/O from how I/O is executed. The "what" is specified by the application programmer in a persistent list that contains the class data members that are available for I/O. How I/O is actually executed is determined by another component, an I/O module. The I/O module knows how to extract persistent locations from objects' MetaTypes and how to perform some type of I/O operation. There may be several different I/O modules in a system, each corresponding to different data formats for the same objects.

For example, GammaLaw class members are

```
REAL                pmin;     // minimum pressure
REAL                gamma;    // adiabatic gamma
```

The persistents are listed outside the class declaration:

```
template< class C >
BEGIN_PERSISTENT( GammaLaw< C > )
    PERSISTENT( REAL, pmin, "pmin" )
    PERSISTENT( REAL, gamma, "gamma" )
END_PERSISTENT
```

Persistents support factorization by separating I/O modules from application modules and by deferring decisions about data initialization until run-time. Factoring I/O out of application objects ensures consistent input and output formatting with essentially no burden on the application programmer, and localizes changes required to support new data formats.

4 Sharing Fields Between Components - DataDirectory

Two models may use different (or the same) fields to compute their respective results. Because we want to use virtual-function polymorphism to call the two models interchangeably, the models must use the same calling sequence in their respective evaluation functions. To avoid passing different fields in the argument list of the evaluation functions, we have developed another alternative: passing a single data structure to the EOS model constructor, which then holds all the fields needed for a material. This data structure is termed the DataDirectory.

The DataDirectory is actually just like any other Tecolote component, except it may have any number of persistents with any names. In contrast, ordinary components may only contain the persistents specified in their persistent lists. Any Tecolote component may be placed inside a DataDirectory, including another DataDirectory. Thus the DataDirectory is hierarchical, much like a

Unix directory structure. Two entries are automatically PUT in a newly created DataDirectory to allow traversal of the hierarchy: Root, which points to the DataDirectory at the top of the hierarchy, and Parent, which points to the immediate predecessor in the hierarchy of the current DataDirectory.

Figure 4 shows the DataDirectory hierarchy used for a multi-material hydro (only a few POOMA Fields are shown for simplicity). Unlike a Unix file directory, however, the DataDirectory has scoping similar to C++ inheritance scoping rules. For instance, when GammaLaw attempts to GET the PhysicsMesh from the Material DataDirectory, it fails to find it. Therefore the search continues by examining the Material Set and Root directories, finding the requested PhysicsMesh in the Root directory. However, while C++ scope is determined by compile-time class inheritance relations, DataDirectory scope is determined by run-time object nesting. “Root.gas.Pressure” is used to explicitly specify the path to the given Field.

The example below, from the GammaLaw class, illustrates the use of the DataDirectory macro GET:

```
ScalarField<C>& IntEnergy(
        GET("IntEnergy", Mat, ScalarField<C>, (Mesh))
);
```

The first argument is the name of the requested item, the second is the DataDirectory in which the search starts (Mat[erial] is a DataDirectory), the third is the type of the DataDirectory item, and the fourth (if present) represents the constructor arguments that are needed if the item is not present and must be added to the DataDirectory.

The DataDirectory improves code factoring because a single data structure is passed to methods that otherwise would use different calling sequences. It defers the association of data with a particular model invocation until the actual invocation, providing generalized parameter-passing without explicit parameter lists. It enhances integration by supplying a mechanism for transparent data sharing among independent modules.

5 Building Components from an Input File - Scripting

At the start of its execution, a program must specify both which modules to use and their initial data values. Tecolote uses a different component for each option and employs persistents to fill in the data needed by the option. Therefore, a Tecolote program built from components also must use a methodology that creates objects from those components and places persistent data in the objects. We chose to incorporate a scripting language into the Tecolote framework to accomplish these tasks.

Each object is described by an object name, a MetaType name, and a list of persistent values. Object hierarchy (or nesting) is indicated by listing one object in the persistent list of another. The following example shows nested objects where a GammaLaw is created as the Eos persistent of a Material. An object’s constructor is called before its persistents are loaded. Therefore an optional initialize function can be called after an object's persistents have been loaded to perform further initialization dependent upon persistent values.

```
gas = Material(
    Eos = GammaLaw(
        gamma = 0.5,
        pmin = 0.001
    )
),
```

The input file describes the initial object hierarchy of a program. In addition, it selects which components will be used and the initial values of their persistents.

In both debugging and actual use, it is desirable to change the control flow of a program without rebuilding the entire code. Tecolote provides this flexibility by allowing the user to specify higher level function sequences in the input file. The necessary facilities are already provided by Tecolote and may be extended by an application programmer through a MetaType that maps methods and functions into function objects that are usable from the input file.

The language, based on Backus' FP language [1] and Robinson's IFP [4], includes program-forming operations (PFO's), basic objects and elementary operations. PFO's include control structures such as branching (If), looping (While), and sequences (Compose). Examples of basic objects are List, number, and string. Elementary operations that act on basic objects might include arithmetic, comparison, and collection functions such as concatenate and length. As in FP, users may define new objects and functions, but not new PFO's.

6 Registering a Component with Tecolote - MetaTypes

In most applications, a module must directly reference other modules with which it interacts. This requirement obstructs factorization and prevents the application programmer from deferring module interactions until run-time. In addition, it is difficult to replace modules that are referenced in multiple locations within the program. In contrast, Tecolote components are registered only once in a table, the MetaSet, that contains all the program components (see Figure 6). Individual components interact indirectly through this table, promoting component independence.

A module is registered as a component with Tecolote by using a MetaType which, when invoked, automatically registers itself with the MetaSet. The MetaType for a class has a name in the MetaSet, holds the persistent list for that class, can create and initialize that class, carries its C++ type information to run-time, and can convert the MetaType from or to a single base class. An object's MetaType is an object in its own right, much like Java's "Class" class [XXX: ref?].

Many languages create a unified type hierarchy by having a single base class from which all classes are derived (Java's Object class, for example). Tecolote classes, on the other hand, do not share a common base class. Relaxing this restriction allows classes not written for the framework (such as STL container classes and C++ basic types) to be incorporated into Tecolote. Non-Tecolote classes are incorporated into the Tecolote type system by describing their basic features and their persistents in a MetaType.

In the example below, the GammaLaw¡Cell¿ class is registered with the framework. The generic MetaTecolote class can create a GammaLaw object only if the GammaLaw class has a constructor taking a DataDirectory* and a String as arguments. A class is said to be Tecolote-aware if it has such a constructor. While a class should be Tecolote-aware to be used to the fullest extent in the framework, this is not a requirement for its use. The class GammaLaw¡Cell¿ is given the name "GammaLaw" in the MetaSet and has the base class Eos. The MAKE_PERSISTENTS macro registers the persistent information defined for the GammaLaw class.

```
#include "GammaLaw.hh"
static MetaTecolote<GammaLaw<Cell>, Eos>
    GammaLawMeta("GammaLaw", MAKE_PERSISTENTS(GammaLaw<Cell>));
```

In addition to providing indirection between model components, the MetaSet separates I/O modules from physics modules in a program: I/O modules have knowledge of other modules only through the MetaSet, while physics modules have no knowledge of I/O modules. This separation enables us to rewrite I/O modules and physics modules completely independently. The independent I/O modules illustrated in Figure 6 are the Parser, which reads an input file to create new component objects, and a Printer, which prints existing component objects to an output file.

The MetaType and MetaSet are key elements for supporting factorization in Tecolote. MetaTypes turn C++ classes into components and the MetaSet may be searched for any component in an application program, providing a level of indirection between components.

7 Conclusion

By using the various techniques in the Tecolote Framework, the Los Alamos National Laboratory's Blanca Project has been able to integrate a wide variety of physics packages into codes with relative ease. We have added new physics models by modifying the code only in the MetaSet and without having to rewrite any of the code's I/O modules. Complex data sharing is accomplished by the DataDirectory, which allows us to avoid complex calling sequences or global variables. By deferring the choice of components until they are specified in the input file at run-time, we ensure maximum flexibility in the code. The combined benefits from this component architecture approach ensure a far simpler and faster method for adding new physics modules into a program.

Future inquiries will be directed towards the effects of Tecolote's component architecture and functional scripting lanugage. Potential areas of investigation include: application programming methodologies in a functional scripting language; the high-level expression of parallelism in the component architecture, including consideration of the effects of the functional nature of the input file language; and the applicability of a funcional language in specifying object hierarchies and application control flow.

References

1. J. Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. In *Communications of the ACM*, pages 613–641, August 1978.
2. K.S. Holian, L.A. Ankeny, S.P. Clancy, J.H. Hall, J.C. Marshall, G.R. Mcnamara, J. W.Painter, and M.E. Zander. TECOLOTE, an Object-Oriented Framework for Hydrodynamics Physics. In *Proceedings of the Conference on Numerical Simulations and Physical Processes Related to Shock Waves in Condensed Media*, Oxford, England, September 1997.
3. J. V. W. Reynders, J. C. Cummings, P. J. Hinker, M. Tholburn, S. Banerjee, M. Srikant, S. Karmesin, Atlas S., K. Keahy, and W. F. Humphrey. *POOMA: A Framework for Scientific Computing Applications on Parallel Architectures.* MIT Press, 1996.
4. A. Robison. Illinois functional programming: A tutorial. *BYTE*, pages 115–125, February 1987.